Journal of King Saud University – Computer and Information Sciences (2016) **28**, 82–97

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

Automatic path-oriented test data generation by boundary hypercuboids

Shahram Moadab [a,*], **Hassan Rashidi** [b]

[a] *Department of Electrical, IT and Computer Sciences, Qazvin Branch, Islamic Azad University, Qazvin, Iran*
[b] *Department of Mathematics and Computer Science, Allameh Tabataba'i University, Tehran, Iran*

Received 6 June 2014; revised 21 March 2015; accepted 13 May 2015
Available online 2 November 2015

KEYWORDS

Path-oriented testing;
Boundary hypercuboid;
Symbolic execution;
Fault detection probability;
Fault detection speed

Abstract Designing test cases and generating data are very important phases in software engineering these days. In order to generate test data, some generators such as random test data generators, data specification generators and path-oriented (Path-Wise) test data generators are employed. One of the most important problems in the path-oriented test data generator is the lack of attention given to discovering faults by the test data. In this paper an approach is proposed to generate some test data automatically so that we can realize the goal of discovering more faults in less time. The number of faults near the boundaries of the input domain is more than the center, according to the Pareto 80–20 principle the test data of this approach will be generated at 20% of the allowable area boundary. To do this, we extracted the boundary hypercuboids and then the test data will be generated by exploiting these hypercuboids. The experimental results show that the fault detection probability and the fault detection speed are improved significantly, compared with the previous approaches. By generating data in this way, more faults are discovered in a short period of time which makes it more possible to deliver products on time.

1. Introduction

The purpose of software testing process is to improve the reliability of the software product (Cai et al., 2008). This process performs a remarkable role in defining and improving the software development process (Hamlet, 1995). The testing process, therefore, attracted special attention as one of the most important stages in the software development. Performing such an important process is only possible with principled design of the test case which refers to the process of identifying program input data so that it satisfies the selected criteria of the testing (Korel, 1990). There are three types of generators, namely: random generators, data specification generators and path-oriented generators (Ryu and Yi, 1999; Nirpal and Kale, 2011). The random generator, which is the most simple test data generator, uses random algorithms to test data generation. The probability that generated test data using this method will be able to satisfy the constraint of the program being tested is very low. The data specification generator

* Corresponding author.
E-mail addresses: moadabsh@gmail.com (S. Moadab), hrashi@atu.ac.ir (H. Rashidi).
Peer review under responsibility of King Saud University.

ELSEVIER Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.jksuci.2015.05.002
1319-1578

selects the test data through program features. This generator generates test data by using some formal methods. The disadvantage of this generator is the software requirement for a formal specification. Generating test data by the path-oriented test data generator is a strong method among the aforementioned approaches (Mansour and Salame, 2004; Zhang et al., 2004). In this approach, test data are generated in order to satisfy the constraints obtained from a traversal path.

Since there is no deterministic solution to satisfy every constraint on the bounded domain (Hentenryck et al., 1998), all previous works are devoted to produce more desirable test data (instead of solving the problem completely). In Offutt et al. (1999), the constraints of the program are obtained after traversing the program path by utilizing the Dynamic Domain Reduction (DDR) process. In the next step, some parts of these constraint allowable areas are eliminated by a search method and by using the split-point method. Finally, appropriate test data are generated from the remaining area. In Zhang et al. (2004), a path-oriented approach is used to solve the problem, which is based on the combination of symbolic execution and constraint analysis. Various methods are discussed to present path conditions and expressions. Chen and Zhong (2008) implemented a multi-population genetic algorithm (MPGA) to support MATLAB. The algorithm selects subjects for free of charge migration based on fitness values. It is a relatively novel and meaningful attempt to use MPGA to produce path-oriented test data generation. The basic process flow of path-oriented test data generation using genetic algorithm (GA) describes how to change path-oriented test data generation into an optimization problem. In the Path-oriented Random Testing (PRT) (Gotlieb and Petit, 2010) proposed by Gotlieb and Petit, random and uniform test data are generated by using a divide-and-conquer algorithm. Generating test data over obtained constraints in the PRT approach is based on two basic concepts: constraint propagation and constraint refutation. The constraint propagation process, prunes the variables domain of inconsistent values by using an iterative algorithm. Then, by using the constraint refutation process, some parts of the non-allowable areas are eliminated. Finally, the related test data are generated from the remaining areas. A path-oriented automatic testing method was proposed by Zhao and Huang, based on double constraint propagation (Zhao and Huang, 2012). The domain of a path can be generated by splitting an input variable domain and executing a double constraint propagation algorithm. Besides, according to the reduced path domain, a random test data generator can be developed.

The main purpose of our proposed approach is to pay serious attention to discover the diagnostic test data, based on spending less time, cost, and effort to obtain the data as well as utilizing fewer resources. There is lots of evidence (Ahamed, 2009; Abreu et al., 2005; Salem et al., 2004) that conveys that there are more faults in the boundaries of the areas. Therefore, this proposed approach tries to discover more faults in the less time by focusing on the boundary domains. This task is performed in six main steps: control flow graph construction, defining the basis for a path set, symbolic execution of a path, constraint propagation, extracting boundary hypercuboids and test data generation. The manual test data generation is time consuming, boring and expensive (Edvardsson et al., 1999), besides fixing the mentioned disadvantages, the automation of this approach is an important issue, to which attention is given in this research.

In Section 2, we illustrate concepts and important terms of test data generation. In Section 3 we explain one of the most important path-oriented test data techniques and its common methods. In Section 4 a new approach is presented to generate path-oriented test data and we evaluate them at a later time. Section 5 contains the experimental results obtained from our implementation. In Sections 6 and 7, we respectively discuss the threats to validity and present the conclusion and proposed cases for future works.

2. Basic concepts

Before explaining the proposed approach, the most important concepts and the basic terms need to be clarified.

2.1. Control flow graph

The control flow graph (CFG) is a directed graph for a program, which presents the relation $G = \{N, E, s, f\}$. In this relation, N is the graph's nodes set, E is the edge set of the graph ($E = \{(n_i, n_{i+1}) | n_i, n_{i+1} \subseteq N\}$), s and f are the entry and exit nodes, in order. Each node represents a linear sequence of computations in the program. Each edge $e = \{n_i, n_j\}$ is a transfer of control from node n_i to node n_j and if its condition is satisfied, will lead to the transfer of control from node n_i to node n_j. If we have a Foo function as a program shown in Fig. 1, then the CFG for this function can be depicted by Fig. 2.

2.2. Infeasible path

A path in a CFG is a sequence of nodes $A = n_{a_1} n_{a_2} \ldots n_{a_{m+1}}$ in which $n_{a_1} = s$ and $n_{a_{m+1}} = f$. Alternatively a path is the sequence of $A = e_{a_1} e_{a_2} \ldots e_{a_m}$ in which m is the length of the path A and $e_{a_i} = \{n_{a_i}, n_{a_{i+1}}\}$ $(0 < i \leqslant m)$. By definition, an infeasible path is a non-traversable path displaying inconsistent conditions. In other words, none of test data are able to satisfy all the infeasible path constraints. Since these paths are not traversable at all, the effort made to generate test data to traverse them is completely useless. In selecting path steps, we should avoid selecting these kinds of paths as much as possible. Because of the amount of the time consumed in discovering these paths, it would be a waste of time to generate test data to traversing the paths.

2.3. Allowable area

A domain that is obtained from these intersections present constraints among a constraint set is called the allowable area. For instance, the hatched area of the Fig. 3 shows the allowable area of the constraints set $\{x \leqslant 32767, y \geqslant 0, x > y\}$.

```
void Foo (int x, int y){
    if (x ≥ 500 && y ≥ 0 && x ≥ y)
        cout << "Yes";
    else
        cout << "No";
}
```

Figure 1 The Foo function.

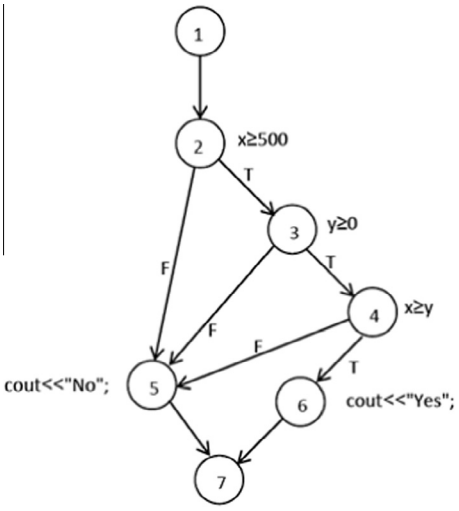

Figure 2 CFG of the function Foo.

2.4. Diagnostic test data

From this point of view, test data are divided into two general groups as depicted in the Fig. 4. Acceptable test data are able to traverse the path, and the rejected test data are data that are not able to traverse the path. Acceptable test data are divided into two groups: some test data which leads to expected output generation and the others which do not lead to expected output generation. By definition, these test data which do not lead to the expected output generation are called diagnostic test data.

2.5. Fault-prone areas

Test data that cause the same path execution do not have the same ability in error detection (Gotlieb and Petit, 2010). This means there is no equal probability of test detection with existing diagnostic test data in different parts of the allowable areas. The part of allowable area that contains at least one diagnostic test data is called a fault-prone area and the entire area set is called fault-prone areas.

3. Hypercuboid

The input domain of the program under testing is formed by the Cartesian product of the integer intervals. These kinds of input domain are called hypercuboids which are an

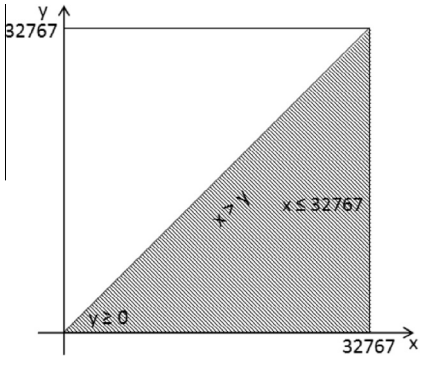

Figure 3 A sample for displaying the allowable area.

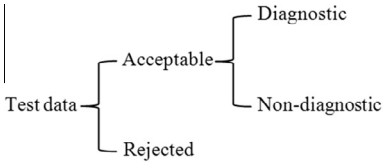

Figure 4 Different types of the test data.

n-dimensional extension of a 3-dimensional cuboid. On a hypercuboid, test data generation is simple to perform because any of its points can be generated by selecting its coordinates independently. For example, in a 2-dimensional (x, y) input space, each of x and y values can be selected and randomized without considering the other one. For instance, consider the Fig. 3. If the constraint $x > y$ is eliminated from its constraint set, and then add the constraints $x \geqslant 0$ and $y \leqslant 32{,}767$, we will have a 2-dimensional hypercuboid. By selecting any points from x and y in the allowable areas (from 0 to 32,767) these constraints are satisfied whereas, by considering the $x > y$ constraint, generating appropriate test data will be a very hard task.

3.1. Classification of hypercuboids

From the perspective of our proposed approach, hypercuboids are divided into four main groups:

- **External (Inconsistent):** It does not have any intersection points with the allowable area; i.e. it is completely outside. None of its generated test data are able to satisfy the entire path constraints.
- **Boundary:** Some of its parts are inside of the allowable area and some other parts are outside. Some of their generated test data are able to satisfy all the path constraints and the others are not able to do this.
- **Internal:** It is completely inside the allowable area and none of its points are exactly on the boundary of the allowable area. The entire generated test data are able to satisfy all the path constraints.
- **Internal-boundary:** The whole of this group is inside the allowable area and at least one of their point places is on the boundary of the allowable area. The entire generated test data of these types are able to satisfy the all path constraints.

Consider the samples of hypercuboids types in the Fig. 5 in which the hatched area shows the allowable area. In this figure, D_1, D_2 and D_5 are external hypercuboids, D_3, D_6, D_7, D_9, D_{10} and D_{13} are boundary hypercuboids, D_{11} is an internal hypercuboid and the rest are internal-boundary hypercuboids.

3.2. Test data generation by hypercuboid

From one view point, the most important work done in the path-oriented test data generation field has been addressed by the following approaches for the test data generation:

- **Eliminating a part of the allowable area:** In this approach, a part of the allowable area eliminates, by a legal method, the partial solving of constraints and appropriate test data generation. DDR method (Offutt et al., 1999) is one of the

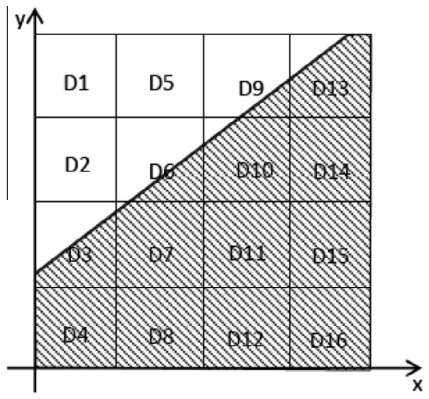

Figure 5 Some samples of the hypercuboids.

most famous works that has been done in this field. In this method, after getting the constraints of a path, a hypercuboid would be obtained by eliminating parts of the allowable area of these constraints. Finally, the proper test data are generated from an obtained hypercuboid. Although the DDR method was partially able to satisfy constraints, it still suffers from some deficiencies. The most important disadvantage of this method is that it is eliminating parts of the allowable area and lacks the ability to cover them completely. Moreover, these eliminated areas are mostly selected from boundary areas and because boundary areas are more fault-prone, generated test data of this method will have less chance to discover faults.

- **Adding a part of the non-allowable area:** In this approach, parts of the non-allowable area are added to the allowable area for appropriate test data generation. The PRT method (Gotlieb and Petit, 2010) is the most important task that has been done in this field. One of the disadvantages of random test data generation is that the distribution of data might not be uniform over all the allowable area. This disadvantage can lead to some existing test data not being generated in a lower density so they do not have a high fault detection capability. So, in order to solve the aforementioned disadvantage the PRT method tries to generate uniform and fairly tested data. Test data generation of the PRT method, has some applications based on the both constraint propagation and constraint refutation process. Constraint propagation process, by using an iterative algorithm, will generate the smallest hypercuboid which includes the whole related constraint allowable area and also some parts of the non-allowable area. For eliminating some parts of non-allowable area, the constraint refutation technique is used. In this technique, every domain of a generated hypercuboid divides into a k subdomain. If the size of a variable domain is not divisible by k, its domain will be spread to the extent that it divides by k. By the obtained subdomains of the Cartesian product from the n input variable, the k^n elements, which are hypercuboid themselves, will be acquired. Some of the obtained hypercuboids are inconsistent with the constraint set, so they will be eliminated by the constraint refutation process. In other words, the constraint refutation process eliminates all the hypercuboids that do not have any intersection points in common with the allowable area. One of the disadvantages of this method is that the running time of the algorithm increases if its division parameter (k) becomes more than the threshold. Spending a lot of time, can cause later delivery and end-user dissatisfaction. The second disadvantage of this method is the domain division by k in only one phase. It causes time consuming consistency checking which takes place in order to the number obtained in the k^n hypercuboid. By gradual domains division (for example dividing into two parts in every phase) a wider area can be eliminated, with less time checking consistency in each phase. The third disadvantage is that the PRT method gives equal emphasis for the test data generation because of uniform distribution, so the test data do not have the equal ability to discover the fault. In other words, only some of the test data are able to discover the fault. With this interpretation, a large amount of generated test data using this method does not cause fault detection. The fourth disadvantage of this method is the increasing variables domain for making them divisible with k. It may add a remarkable and vain area to the allowable area. The size of this vain area will exponentially increase according to increasing the number of variables.

4. The proposed approach

Many of the previous works (Alzabidi et al., 2009; Swathi et al., 2011; Catelani et al., 2011; Gong et al., 2011; Boghdady et al., 2011) dealt with only test data generation and did not pay too much attention to the ability of the generated test data in the fault detection, while some appropriate test data can lead to fault detection (Myers et al., 2004). Of course, discovering every fault is not possible, especially when delivering the first version of the software product. The main goal is to deliver the software products with appropriate reliability in a limited time. Time delay is one of the most important factors which causes software project failure. On the other hand, one of the most important factors for end-user satisfaction is being on time. In other words, on time delivery of the product with some errors (but not critical errors) attracts satisfaction more than the late delivery with fewer errors. So, as a solution, instead of discovering all the extractable faults, more and important faults can be found in the expected time interval and then while the end-user is using the product, the rest of the less important faults can be discovered. Since the software testing phase spends about 40–70% of effort, time and cost to itself (Salem et al., 2004; Kosindrdecha and Daengdej, 2010; Xiao et al., 2007) this solution can be counted on as an effective solution to reduce this percentage amount.

Because faults are not uniformly distributed across a portion of any program, some of its parts (based on Pareto 80–20 principle, 20% of programs) are more fault-prone (based on Pareto 80–20 principle 80% faults). The question, here, is that which parts of a program are the most fault-prone and the number of faults in each path in which parts of the allowable area of that path is greater. There is a lot of evidence to prove this important point, so that boundary areas contain more and important faults rather than the central. The following statements by Pressman confirm this focus (Pressman et al., 2003):

- For reasons that are not completely obvious, the existing errors at the boundaries of the input domain are more than in the center.

- All the softwares usually fail at the boundaries.
- In some situations (the most occurs in mathematical algorithms), a small range of existing errors at the valid data boundary of a program may cause extreme and even incorrect processing or deep performance degradation.
- Software engineers often make mistakes at the boundaries of a problem.

Accordingly, many works were devoted to boundary value testing (BVT). There are some existing methods on the boundary value testing, including boundary value analysis (BVA), robustness testing, worst case testing and robustness worst case testing. In order to reduce the opportunity for coincidental correctness, Hierons in Hierons (2006) has shown how boundary value analysis can be adapted. This research demonstrates if only a geometric approach is considered to produce the test input for BVA, we cannot generate any possible input to detect boundary shifts. Chen et al. (2007) proposed an approach based on the concept of virtual images of the successful test cases. This approach relies on a rule that test case generation should refer to the locations of successful test cases (those that do not reveal failures). This rule ensures that all test cases are far apart and evenly spread in the input domain, because more test cases are generated near the boundary of the input domain. This research analyzed the cause of this boundary effects and their implementation based on the concept. Zhao and Li (2009) also present a novel boundary value testing approach based on the principle of fault diagnosis in integrated circuits. In this research a new boundary test case selection is generated by applying fault detection rules in the circuits. Moreover, a corresponding automatic test cases generation tool is developed and based on a real-world application, authors designed three boundary test suites.

These works were devoted to BVT for exact points on the boundary or attached to the boundary; not the principle of the area around the boundary. On the other hand, at every stage of the life cycle of any software product, a small percentage of files will contain a large percentage of the faults detected (Ostrand and Weyuker, 2002). In this regard, one of the important principles of testing is as follows: during the testing process in any software product for 20% of all the components, 80% of all the discovered faults are likely traceable (Pressman et al., 2003). This principle is relied on the Pareto 80–20 principle. The Pareto's principle was coined by Vilfredo Pareto (Italian economist) (Pareto, 1906). In 1906 he observed that 20% of the Italian population owned 80% of property in Italy. In the late 1940s management leader, Dr. Joseph M. Juran suggested that a small number of causes define most of the results in any cases (Juran, 1951). Hence, Pareto's rule or the Pareto 80–20 principle was born and then has been attributed to the economist and applied to many sciences such as the treatment of defects in software (Boehm and Basili, 2001). By studying different researches on reliability and use of Pareto's principle in computer science (Iqbal and Rizwan, 2009; Rizwan and Iqbal, 2011; Lipovetsky, 2009; Pandey et al., 2013) as well as in software testing and debugging (Kuo and Huang, 2010; Kuo et al., 2012; Andersson and Runeson, 2007; Xiangyun and Wenjing, 2011; Fenton and Ohlsson, 2000; Zhang, 2008), we can conclude that although the 20% is not an exact number but approximately around 80% of defects occur in 20% of code (Gittens et al., 2005). Considering the above knowledge on boundary faults and Pareto 80–20 principle, we can conclude that the efforts to generate test data at the 20% of boundary (instead of the entire area), usually lead to some achievements on the expected reliability and end-user satisfaction.

According to the explanation mentioned, efforts have been performed to generate test data at 20% of the allowable area boundary in the proposed approach. It must be denoted that the amount of 20% regarding to the time constraints on any software product can be changed from 0% to 100% (see Section 5.1.). After delivering the product to the end-user, we will be able to discover more faults, because of less limited time, by increasing it to 20% and covering more allowable area.

4.1. Architecture of the proposed approach

Before starting a complete elaboration on the proposed approach, it would be better to show the stages in detail. The architecture drawn in the Fig. 6 shows these stages. As the proposed architecture shows, our approach applies in six basic stages. Each of the above six stages and their complete details will be explained completely in the following subsections.

4.1.1. Control flow graph construction

The first step of the proposed approach is to construct the control flow graph. To perform this, at first the program traverses and then the related control flow graph will be drawn based on the stated explanation in the Section 2.1.

4.1.2. Basis path set

After CFG construction, some of its path must be selected for testing. In our approach, these paths are selected based on McCabes' proposed test (McCabe, 1976) as the basis path testing. In this test, after calculating the related CFG Cyclomatic complexity and applying test adequacy criteria (Mansour and Salame, 2004; Swathi et al., 2011; Myers et al., 2004; Ehmer-Khan, 2011), the basis path set, including linearly independent paths, are constructed.

4.1.3. Symbolic execution of a path

In order to create some algebraic expressions for the constraints in programs, symbolic techniques assign symbolic values to variables. These techniques use some constraint solver to find a solution for the expressions that satisfies a test requirement. The execution flow is performed in the normal direction except that values may be symbolic formulas over the input symbols (King, 1976). The main goal of the symbolic execution is to extract a set of constraints C_1 to C_n on a path which is shown by $C_1 \hat{} \ldots \hat{} C_n$.

In this step, a path is selected from the previous basis path set, and its constraint set will be obtained by the symbolic execution. It should be noted that after applying all the next stages on this path and finally its test data generation, the next path is selected and all the above actions on it will be performed. This continues until selecting all the existing paths of the basis path. If all the paths were selected and there are no paths to be selected, the algorithm ends.

4.1.4. Constraint propagation

Any constraints on the values of variables involved in the constraints can be effective. For example, suppose the minimum

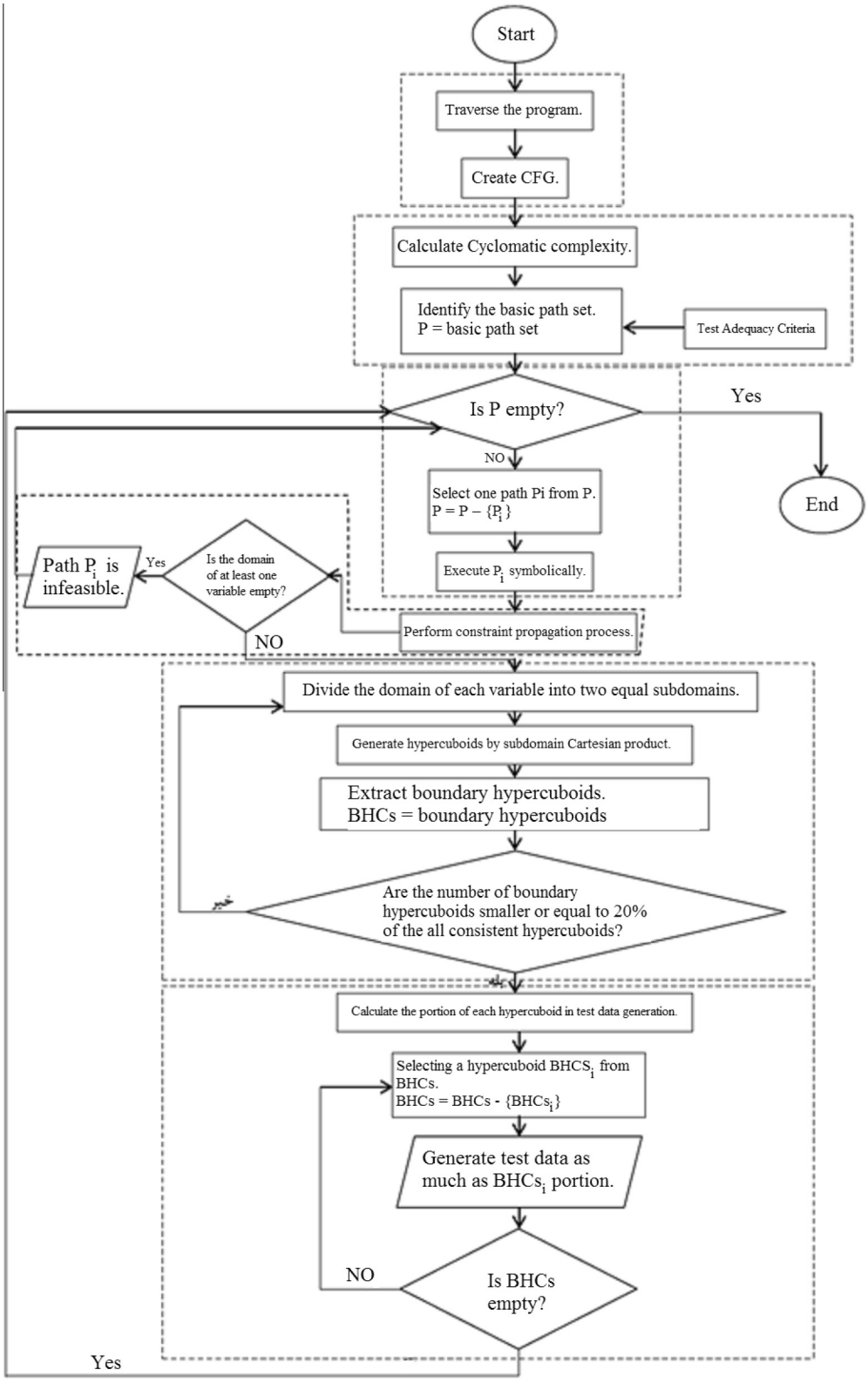

Figure 6 The architecture of the proposed Approach.

and maximum range of the variables x and y, respectively, are $0 \leqslant x \leqslant 4000$ and $1000 \leqslant y \leqslant 3000$. Considering the constraint $x \geqslant y$, the minimum y (i.e. 1000), prunes the minimum value of x to 1000. However, the maximum value of x (i.e. 4000) cannot prune y to 4000, because it is bigger than the maximum value of y (i.e. 3000). Consequently, the minimum and maximum ranges for the variables x and y will be $1000 \leqslant x \leqslant 4000$ and $1000 \leqslant y \leqslant 3000$, respectively. The pruning operations are

repeated as long as there are more possible prunes on the variables (Gotlieb and Petit, 2010).

The result of the constraint propagation process will be a range between the minimum and maximum of each variable. Intersection of these ranges constructs a hypercuboid. The obtained hypercuboid is the smallest that contains all the ranges of the variables. If the range of at least one variable equals to empty, the intersection of all constraints will be empty, too. It means that none of the test data are able to satisfy the related path and the path is infeasible. So, the algorithm returns to the previous stage which is selecting the next path and its symbolic execution. The most frequently repeated pruning operation is performed when the selected-path becomes an infeasible path (see Section 5.4.).

4.1.5. Boundary hypercuboid extraction

In this step, in the beginning, each input hypercuboid divides into sub-hypercuboids at the beginning. Each of these sub hypercuboids is also a hypercuboid. Boundary hypercuboids are extracted among these obtained hypercuboids. This process goes on until the maximum 20% boundary hypercuboid is achieved.

To extract hypercuboids, at first all the hypercuboid variable domains are divided into two equal subdomains and then new hypercuboids are obtained from the obtained subdomain Cartesian product. For instance, hypercuboids D1 to D4 are obtained from Cartesian product of the Fig. 7 subdomains that are shown in Fig. 8.

After extracting these hypercuboids, boundary hypercuboids are extracted and since internal-boundary hypercuboids are around the allowable area, they are considered as the boundary hypercuboids. The consistency checking algorithm is used to extract boundary hypercuboids. Since one of the most time consuming parts of the PRT algorithm was its consistency checking algorithm, we presented a new consistency checking algorithm to reduce the running time of this algorithm. It is obvious that if a hypercuboid has an n input variable, it will have 2^n corners. In the consistency checking algorithm, if none of the hypercuboids corners are placed at the allowable area, it would be eliminated as an external hypercuboid. If the numbers of corners exist inside the allowable area and some outside of it, the related hypercuboid would definitely be a boundary. Also, if all the corners have been inside the allowable area and some outside of it, the hypercuboid is considered as an external hypercuboid if at least one of the internal hypercuboids corners was placed exactly

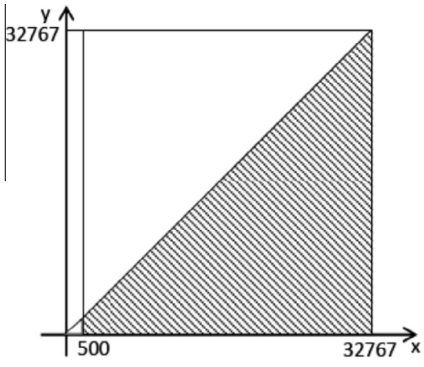

Figure 7 Demonstration of an allowable area.

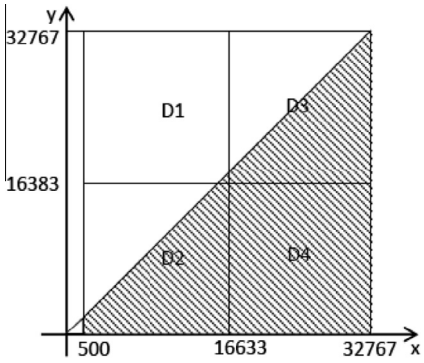

Figure 8 Demonstration of extracting sub-hypercuboid.

on the allowable boundary, it is considered as internal-boundary. In other words, if the entry conditions equal PC_0, PC_1 ... and PC_{m-1} and also the equality conditions mode (exactly the boundary values) equal PH_0, PH_1 ... and PH_{m-1} then the hypercuboid will have one of these statuses:

- **It is external:** if we have this for all its corners:

  not PC_0 || not PC_1 || ... || not PC_{m-1} = not (PC_0 && PC_1 && ... && PC_{m-1}).

- **It is internal:** if we have this for all its corners:

  (PC_0 && PC_1 && ... && PC_{m-1}) && (not PH_0 && not PH_1 && ... && not PH_{m-1}) =(PC_0 && PC_1 && ... && PC_{m-1}) && (not (PH_0 || PH_1 || ... || PH_{m-1}))

- **It is boundary or internal-boundary:** if it does not to be external or internal.

Although the proposed algorithm does not work properly in few cases, it is very useful in extreme time improvement. For instance, in a rare case, some angles of the allowable area corners might be low. In this case, all the corners of a sub-hypercuboid might be out of the allowable area and some of them placed inside of it. Although the hypercuboid is a boundary, it is eliminated as an external hypercuboid. The hypercuboid D9 in Fig. 9 is an example of it. Also, all the corners

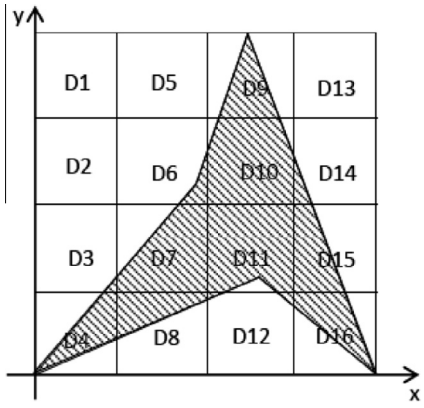

Figure 9 A demonstration of the consistency checking algorithm problem.

of the hypercuboid might be placed completely in the allowable area and some of its internal area is placed outside of it. In this case the hypercuboid is eliminated as an internal hypercuboid. The hypercuboid D11 in the Fig. 9 is a sample of this.

The above problem is not counted as a big challenge for two reasons. First, it would not occur in many programs. Second, if it happens, only a few hypercuboids will be lost and the test data can be generated from other boundary hypercuboids. It should be noted that the consistency checking algorithm would never make a mistake in recognizing boundary hypercuboids. Because if some of the hypercuboid corners are inside the allowable area and some are outside of it, then the related hypercuboid would be the boundary. So, a big time improvement regarding this algorithm is preferred over its small problem. In the end, if the numbers of the boundary hypercuboids became less than or equal to 20% of the total boundary and internal hypercuboids (all the consistent hypercuboids), the algorithm returns to the next phase. Otherwise, boundary hypercuboids extraction phases will be performed again on newly obtained hypercuboids.

4.1.6. Test data generation

At the last phase of our method, the portion of each hypercuboid calculated in the test data generation is in the first place. For instance, if we need to generate 5000 test data from 100 boundary hypercuboids, the portion of each hypercuboid will be equal to $\frac{5000}{100} = 50$ test data. Finally, a boundary hypercuboid is selected, one after another, and the related test data generate as the calculated count. If the number of requested test data is not divisible by the number of boundary hypercuboids, all the hypercuboids, except the last one, will be given an equal portion and the last hypercuboid gets the rest of the requested portion.

The proposed algorithm, deals with the next path of the symbolic execution after generating test data for the selected path and it then deals with redoing the third to the sixth steps along the path. This algorithm terminates if all of the paths were chosen and there are no more unselected paths.

4.2. Boundary Path-oriented Random Testing (BPRT) proposed algorithm

The proposed algorithm has been implemented by the BPRT algorithm and this is shown in Table 1. This algorithm has been implemented by a divide-and-conquer technique. In the BPRT algorithm, variable V is a one-dimensional dynamic array that keeps the names of variables of the path conditions. BHCs is a 3-dimensional dynamic array that keeps boundary hypercuboids within itself. PC is a dynamic array which contains the related path conditions that are obtained by the symbolic execution process. PH is a dynamic array that stores the equality conditions mode of the related path within itself. For instance, the equality mode of condition $x > y$ would be $x == y$. HCs is a 3-dimensional dynamic array that keeps generated hypercuboids from the Cartesian product of the boundary hypercuboids subdomains within itself.

First, the operational purpose of the BPRT algorithm is to extract the boundary hypercuboids by using the Cartesian and Consistent functions. While the proportion of the boundary hypercuboids to the total consistent hypercuboids (i.e. BHC/(BHC + IHC)) is more than 0.2, extracting the boundary

Table 1 The BPRT proposed algorithm.

```
BPRT(V[VarCnt], BHCs[BHC][2][VarCnt], PC[m], PH[m], N){
  b = 0.2;
  while (BHC/(BHC + IHC) > b){
    HCs[BHC * pow(2, VarCnt)][2][VarCnt]
      = Cartesian(V[VarCnt], BHCs[BHC][2][VarCnt];
    BHCs[BHC][2][VarCnt] = Cansistent(HCs[BHC
      * pow(2, VarCnt)][2][VarCnt], PC[m], PH[m];
  }/*End of while*/
  N1 = floor(N/BHC);
  N2 = N – N1 * (BHC – 1);
  for (i = 0; i < BHC; i++){
    if (i < BHC – 1)
      TN = N1;
    else
      TN = N2;
    while (TN > 0){
      Pick up t uniformly at random from BHC[i][2][VarCnt];
      if (PC is satisfied by t){
        add t to T;
        TN = TN – 1;
      }/*End of if*/
    }/*End of while*/
  }/*End of for*/
  return T;
}
```

hypercuboids is repeated utilizing these two functions. In the Cartesian function, the minimum and maximum domain of each variable is divided into two equal subdomains and a hypercuboid is obtained from one of the two subdomains of all the variables of the Cartesian product. It means that one of the first subdomains or the second one is selected from the first variable, one of the two subdomains is selected from the second variable, too and ... one of the subdomains is selected from the variable VarCnt in this order. The selective set constructs a new hypercuboid. Finally, generated hypercuboids are returned as a result by the array HCs. In Consistent function after extracting all the boundary hypercuboids according to the mentioned definitions in the Section 4.1.5 and storing them in the BHCs array, this array will be returned as a result. After extracting boundary hypercuboids, all their portions for test data generation will be calculated. Finally, after selecting the first hypercuboid, test data will be generated randomly by the random function of Turbo C++. This process repeats until selecting the last hypercuboid and its test data are generated (it means BHC times). The generated test data set are returned by the variable T.

4.3. Evaluation and comparison of the proposed BPRT algorithm

To present any algorithms, the evaluation of it, is one of the most important things to do. In this Section, in addition to evaluating the BPRT algorithm, it will be compared to the previous algorithms. Before this, the criterion of evaluation is described.

4.3.1. Evaluation criteria

In the PRT method, one of the criteria is the sum of the test data rejected. Though, it is a good criterion, it is not enough.

Table 2 Experiments results on the Pareto 80–20 principle.

Boundary %		10%	20%	30%	40%
Foo	E	341	343	186	111
	R	347	327	479	558
	Ts	4.11	3.73	3.68	3.74
	F	***0.253***	***0.258***	***0.126***	***0.071***
	S	***82.97***	***91.96***	***50.54***	***29.68***
Triangle	E	864	837	832	837
	R	482	508	563	635
	Ts	4.67	4.17	4.23	4.28
	F	***0.583***	***0.555***	***0.532***	***0.512***
	S	***185.01***	***200.72***	***196.69***	***195.56***
Tcas	E	461	490	421	367
	R	4259	4516	5016	5481
	Ts	29.5	29.3	30.5	36.8
	F	***0.088***	***0.089***	***0.07***	***0.057***
	S	***15.63***	***16.72***	***13.8***	***9.97***

Because as mentioned before, software testing that does not lead to fault detection cannot be a suitable test. Therefore, we present a fault detection probability criterion based on the following equation.

$$\text{Fault detection probability} = \frac{\text{the number of diagnostic test data}}{\text{the number of the total generated test data}}$$

This criterion defines the fault detection probability of a test data. The next criterion, called fault detection speed, is defined according to the following equation.

$$\text{Fault detection speed} = \frac{\text{the number of diagnostic test data}}{\text{running time}}$$

The number of test data rejected and consistency checking algorithm run time are two effective run time factors. Measuring the run time of the consistency checking algorithm needs a real execution and is not assessable by mathematical equations, so we will measure this parameter with a real execution in the Section 5. The number of the consistency checking algorithm usages can be used for evaluation.

4.3.2. BPRT method evaluation

By relying on the Pareto 80–20 principle, we accept that 80% of faults lie on 20% of boundary allowable area. Thus generating the total generatable test data from 20% boundary of the allowable area, allows to generate 20% of total generatable test data. These test data will be able to detect 80% of faults. In this aspect, we can determine the probability of fault discovered by the test data generated by BPRT method from the boundary hypercuboids (20% of the total of hypercuboids). In other words, for the BPRT method what the fault detection probability (FDP) is. If all the possible test data are generated with the BPRT method and we name the total faults E, the number of test data able to be generated from total allowable hypercuboids N, the fault detection probability, can then be calculated using the following equation according to the Pareto 80–20 principle:

$$\begin{matrix} 20\%N & 80\%E \\ 1 & EDP(BPRT) \end{matrix} \rightarrow \text{BPRT fault detection probability} = \frac{80\%E}{20\%N} = \frac{4E}{N}$$

Calculating the exact number of consistency checking algorithm usages depends on the shape of allowable area. But in a fair state, half of the main area (the obtained area from the constraint propagation execution process) can be assumed as external area and the other half as internal. So, the total usages of this algorithm will approximately be equal to:

The total usages of BPRT consistency checking algorithm $\approx$ BHC + C

Constant C shows the total usages of this algorithm with external and internal areas, which is a small number.

4.3.3. Comparison of BPRT and PRT methods

If all the possible test data are generated by the PRT method and we name the total faults E, the number of test data able to be generated gained from the allowable hypercuboid N, then all the E faults can be discovered using the N test data. Therefore, the fault detection probability is equal to:

$$\text{PRT Fault detection probability} = \frac{E}{N}$$

The obtained BPRT fault detection probability was equal to $\frac{4E}{N}$, so the chance of fault detection, in the BPRT method generated test data would be 4 times higher than the PRT method. The total usages of the consistency checking algorithm in PRT method, by assuming equal and fair situations, in the BPRT method are equal to:

The total usages of PRT consistency checking algorithm = BHC + IHC + EHC

As we saw, this amount was obtained about BHC + C for the BPRT method. Whereas IHC + EHC amount is greater than small constant C, so that BPRT method works better than PRT, too. From another perspective, the BPRT method has solved the considered PRT defects in the Section 3.2. In the BPRT method, consistency checking only takes place by using the corners of the hypercuboids. This leads to extreme running time reduction for the time consuming consistency checking algorithm. Because the first defect of the PRT method is time consuming, running the consistency checking algorithm and subsequently the lack of severe increase in the possibility of division by parameter k is solved. Less use of the consistency checking algorithm by the BPRT method explains solving the second defect of the PRT method, which is domain division using one step. Since the fault detection probability of the generated test data is four times more than the PRT method, the third PRT defect, not playing the diagnostic test data down, has been eliminated using the proposed method. There is also no need to extend the domain of variables to make them divisible by the k parameter in the BPRT method. Therefore, by increasing the variable domains, the last defect of the PRT method has been solved using the BPRT method.

5. Results

To evaluate the proposed approach more precisely, some experiments have been done on both the RT and PRT proposed methods as well as the BPRT method. To be fair, the same situations have been applied for the implementation of all these methods. All these algorithms will be implemented by Turbo C++ 4.5 programing language on a Pentium IV personal computer with 1.6 GHz CPU and 288 MB RAM

Table 3 Experiments results of the function Foo with $N \geqslant 1000$.

N		1000	2000	3000	4000	5000	6000	7000	8000	9000	10,000
RT	E	31	56	91	118	170	179	211	247	269	313
	R	7114	14,813	22,891	29,533	37,764	45,314	51,673	67,319	72,321	76,035
	Ts	17.81	22.11	28.99	32.71	35.34	38.91	41.97	43.24	46.19	49.13
	F	***0.004***	***0.003***	***0.004***	***0.004***	***0.004***	***0.003***	***0.004***	***0.003***	***0.003***	***0.004***
	S	***1.74***	***2.53***	***3.14***	***3.61***	***4.81***	***4.6***	***5.03***	***5.71***	***5.82***	***6.37***
PRT	E	32	58	91	121	161	186	215	255	271	309
	R	755	1659	2487	3231	4279	4981	5744	6277	7414	7953
	Ts	3.62	6.37	6.75	6.82	6.95	7.49	8.09	8.13	9.1	9.39
	F	***0.018***	***0.016***	***0.017***	***0.017***	***0.017***	***0.017***	***0.017***	***0.018***	***0.017***	***0.017***
	S	***8.84***	***9.11***	***13.48***	***17.74***	***23.17***	***24.83***	***26.58***	***31.37***	***29.78***	***32.91***
BPRT	E	343	673	1007	1308	1735	1941	2325	2670	2987	3352
	R	327	660	1043	1349	1699	2020	2297	2707	3088	3228
	Ts	3.73	5.65	6.48	6.81	7.42	7.82	8.36	8.79	9.03	9.39
	F	***0.258***	***0.253***	***0.249***	***0.245***	***0.259***	***0.242***	***0.25***	***0.249***	***0.247***	***0.253***
	S	***91.96***	***119.12***	***155.4***	***192.07***	***233.83***	***248.21***	***278.11***	***303.75***	***330.79***	***356.98***

(a 256 MB RAM with a 32 MB RAM). It should be noted that the best division parameter value will be considered for PRT.

The mentioned algorithms will be tested on five programs. The first program being tested is Foo function which contains faults around the allowable area boundary. The second program is Triangle which has been proposed by Myers et al. (2004). This program contains arithmetic overflow errors. The third program is Middle as the one used in Beyer et al. (2007). This program contains a common programing error in which a pair of braces is missing. The next program being tested is an aircraft collision avoidance system called Tcas. This program has been selected from the Siemens test suite Hutchins et al. (1994). The Siemens programs were assembled by Tom Ostrand et al. at Siemens Corporate Research for the purpose of studying the fault detection capabilities of control-flow and data-flow coverage criteria. Tcas is made up of 173 lines of C code and has 41 faulty versions which can be downloaded from Software-artifact Infrastructure Repository (Dwyer et al., 2012). Different versions of Tcas contain variable errors that have been seeded there by expertise. The last program also being tested is Totinfo which has been selected from the Siemens test suite. Totinfo reads a large number of numeric data tables as input and computes the given statistical input data. It also computes statistics for each table as well as across all tables. This program is made up of 406 lines of C code and has a 23 faulty version which also can be downloaded from Software-artifact Infrastructure Repository. Before performing experiments on these programs, some experiments have been conducted to further evaluate the Pareto 80–20 principle.

In theory, picking bigger values for k would result in increasing some deductions as many additional subdomains will be tested for satisfiability and probably discarded in the PRT method. But the time needed to check satisfiability will also increase accordingly. In practice, selecting small values for k (e.g. k in 1 ... 4) allows to maximize the benefit by eliminating large subdomains while having an acceptable overhead (Gotlieb and Petit, 2010). In order to get better results, the parameter k in the PRT method for function Foo is considered the value 3. The value of this parameter for the Triangle, Middle, Tcas and Totinfo is 4, 4, 1 and 3, respectively.

It should be noted that in all experiments, the stages Control Flow Graph Construction, Basis Path Set and Symbolic Execution of a path have been done manually. The output of these stages is same as the input of the algorithm in the next stage (Constraint Propagation) and other stages are performed by implemented algorithms.

The abbreviation N represents the total number of the acceptable test data generation, E shows the number of diagnostic test data, T represents running time that measured in seconds, R represents the number of test data rejected, F represents the fault detection probability and S shows the fault detection speed. Recall that the values of these variables are equal to $F = \frac{E}{N+R}$ and $S = \frac{E}{T}$.

5.1. Experiments results on the Pareto 80–20 principle

In order to do more investigations on the Pareto 80–20 rule, we did extra experiments with 20% of the boundary. To do this, the experiments on the boundary 10–40% with the step interval 10% were performed. The results of the experiments on three functions Foo, Triangle and Tcas are shown in Table 2. In all experiments, we choose $N = 1000$. Since the fault detection probability (F) and the fault detection speed (S) are used to make the comparisons with other methods in the experimental results, we bold the rows F and S in Tables 2–7.

The results of experiments show that the number 20% is not entirely accurate, but its accuracy is high. For example, in the program Triangle, the maximum fault detection probability at 10% of the boundary ($F = 0.583$) is observed. However, in most cases the best value is 20% and it is greatly reliable. Further experiments need to be performed in this regard.

5.2. Experiments results on function Foo

One of the common mistakes of programing is using the variables of a condition instead of each other. The Foo function shows this mistake in the Fig. 1. The main condition of "if" was ($x \geqslant 0$ && $y \geqslant 500$ && $x \geqslant y$) that was written ($x \geqslant 500$ && $y \geqslant 0$ && $x \geqslant y$) by mistake. To test this

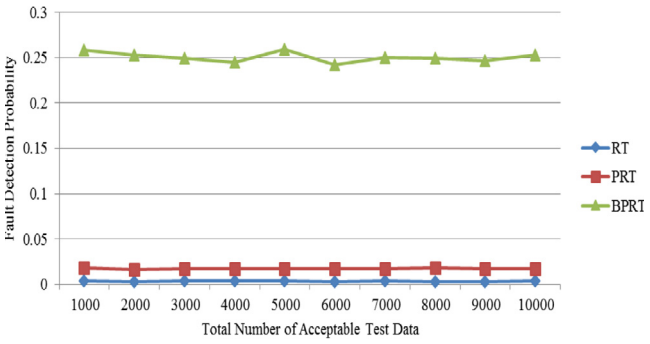

Figure 10 Fault detection probability chart of the function Foo with $N \geqslant 1000$.

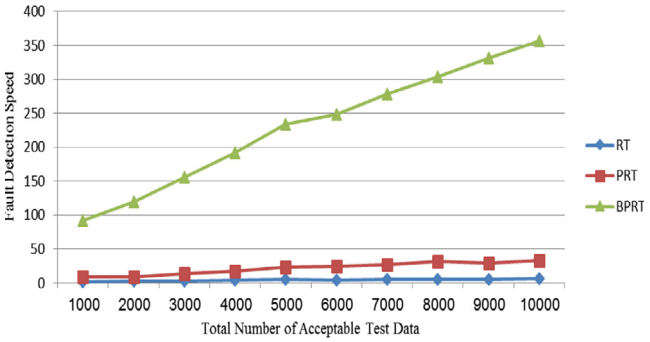

Figure 11 Fault detection speed chart of the function Foo with $N \geqslant 1000$.

program, the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7$ has been selected. Consider the experiment results of the RT, PRT and BPRT methods on the path mentioned in Table 3. These results have been collected by running the algorithms for $N = 1000$–$10{,}000$ with the step interval 1000.

It is not usually possible to obtain precise 20% of the boundary hypercuboids, because the area ranges of hypercuboids are divided into two sub-ranges every time and the "BHC/(BHC + IHC)" will be generally less than the exact amount of 20% in the last time. In this experiment, the BPRT method starts to generate test data after extracting about 12% boundary hypercuboids. To be more precise, 252 boundary hypercuboids, 1891 internal hypercuboids, 1953 external hypercuboids and in total 4096 hypercuboids were determined in this experiment. Also, the length of boundary hypercuboids in the x direction is 504 and its width in the y direction is 512. As an example, for the first boundary hypercuboid extracted, variable x is changing from 500 to 1003 and the variable y is changing from 0 to 511.

Figs. 10 and 11 show the fault detection probability of the aforementioned methods and their fault detection speed based on the total number of acceptable test data, respectively.

Observing the results of these experimental methods on function Foo, the conclusion can be made that the fault detection probability of the BPRT method is about 70 and 15 times more than that of the RT and PRT methods, respectively. It is also observed that the fault detection speed of the BPRT method is about 53 and 11 times more than that of the RT and PRT methods, respectively. Consequently, fault detection speed of the BPRT method increases significantly with a greater amount of test data.

5.3. Experiments results on program triangle

Arithmetic overflow error is another common error in programing (Ahamed, 2009). The Triangle program that was written by Myers contains these kinds of errors. This program checks the possibility of constructing a Triangle by three input numbers. To test this program, the path that contains the main conditions is considered. Consider the experiment results of the related methods on this program in Table 4. These results were obtained by running the algorithms for $N = 1000$–$10{,}000$ with the step interval 1000.

The fault detection probability of the before mentioned methods and their fault detection speed based on the total number of acceptable test data are illustrated in Figs. 12 and 13, respectively.

With the experiment results of these methods about the program Triangle, a conclusion can be made that the fault

Table 4 Experiments results of the program Triangle with $N \geqslant 1000$.

N		1000	2000	3000	4000	5000	6000	7000	8000	9000	10,000
RT	E	537	1096	1661	2314	2696	3415	3898	4443	5005	5521
	R	15,871	30,728	47,163	62,669	76,061	92,016	108,020	122,253	141,480	157,252
	Ts	34.21	62.17	70.95	73.92	76.55	85.12	92.53	98.12	103.29	107.73
	F	***0.032***	***0.033***	***0.033***	***0.035***	***0.033***	***0.035***	***0.034***	***0.034***	***0.033***	***0.033***
	S	***15.7***	***17.63***	***23.41***	***31.3***	***35.22***	***40.12***	***42.13***	***45.28***	***48.46***	***51.25***
PRT	E	559	1110	1665	2234	2746	3313	3818	4432	4947	5519
	R	930	1791	3017	3712	4645	5770	6525	7669	8605	9269
	Ts	3.68	6.26	6.92	7.36	7.64	8.4	9.17	9.66	10.17	10.63
	F	***0.29***	***0.293***	***0.277***	***0.29***	***0.285***	***0.281***	***0.282***	***0.283***	***0.281***	***0.286***
	S	***151.9***	***177.32***	***240.61***	***303.53***	***359.42***	***394.4***	***416.36***	***458.8***	***486.43***	***519.19***
BPRT	E	837	1638	2511	3378	4196	5027	5830	6612	7502	8343
	R	508	1046	1545	2105	2635	3297	3543	4115	4701	5193
	Ts	4.17	5.77	6.32	7.19	7.8	8.3	8.9	9.5	10.11	10.38
	F	***0.555***	***0.538***	***0.552***	***0.553***	***0.55***	***0.541***	***0.553***	***0.546***	***0.548***	***0.549***
	S	***200.72***	***283.88***	***397.31***	***469.82***	***537.95***	***605.66***	***655.06***	***696***	***742.04***	***803.76***

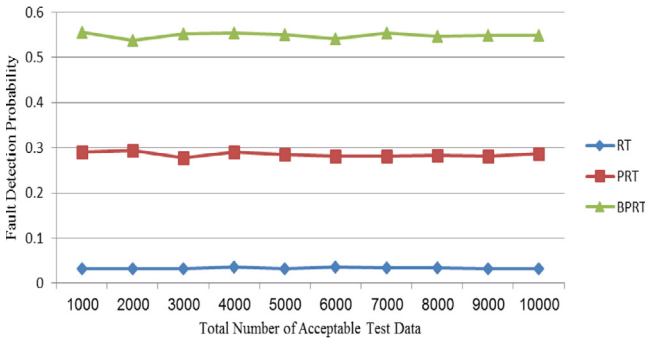

Figure 12 Fault detection probability chart of the program Triangle with N ⩾ 1000.

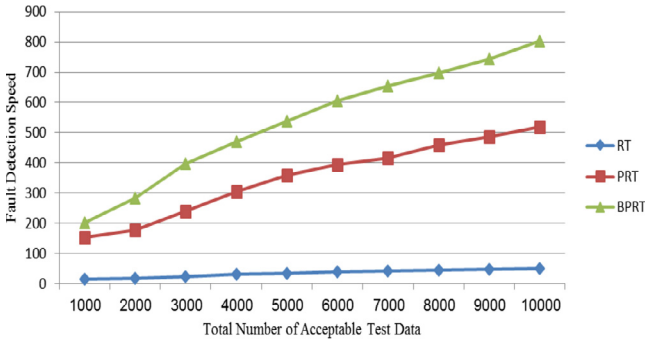

Figure 13 Fault detection speed chart of the program Triangle with $N \geqslant 1000$.

detection probability of the BPRT method is about 16 and 2 times more than that of the RT and PRT methods, respectively. It is also observed that fault detection speed of the BPRT method is about 15 and 1.5 times more than that of the RT and PRT methods, respectively. Moreover, fault detection speed of the BPRT method overcomes that of the other two methods, by using a greater amount of test data.

5.4. Experiments results on program middle

Another common programing error is taken place when the programer forgets to put a pair of brackets in his/her programs. This error could be a very difficult task, because such programs produce correct results for many inputs and only for some inputs produce incorrect outputs. The program Middle contains this kind of error. This program must calculate the mean of three integer values. The program accepts three inputs and invokes the function Middle to calculate the mean. In the experimentation on this program, six possible paths (according to the Cyclomatic complexity) were selected. The two instructions before the *return* are not reachable in the function Middle. So, two paths containing these instructions are infeasible. These paths are found by running the prune operation 32,768 times and then they are removed. Consider the experimental results of the related methods on the four paths for this program in Table 5. These results have been collected by running the algorithms for $N = 1000$–10,000 with the step interval 1000.

Figs. 14 and 15 show the fault detection probability of the aforementioned methods and their fault detection speed based on the total number of acceptable test data, respectively.

Observing the results of these experimental methods on function Middle, the conclusion can be made that the fault detection probability of the BPRT method is about 14 and 1.5 times more than that of the RT and PRT methods, respectively. It is also observed that the fault detection speed of the BPRT method is about 11 and 1.4 times more than that of the RT and PRT methods, respectively. Moreover, fault detection speed of the BPRT and PRT methods increases significantly with a greater amount of test data.

5.5. Experiments results on program Tcas

Tcas program is constructed of 9 functions and 14 global integer variables. A path from the alt-sep-test function is selected to perform the testing. The selected path covers most of the existing conditions. This path will call 4 more after getting the alt-sep-test function called, from which 2 functions of those 4 call functions will call another 6 functions. Consider the experimental results on this path in Table 6. Due to the time consuming experiments regarding this path, the results have been recorded for $N = 100$–1000 with the step interval 100.

The fault detection probability of the before mentioned methods and their fault detection speed by the way of the total

Table 5 Experiments results of the program Middle with $N \geqslant 1000$.

N		1000	2000	3000	4000	5000	6000	7000	8000	9000	10,000
RT	E	487	1019	1427	2025	2515	3311	3486	3895	4526	5256
	R	16,315	31,755	48,222	61,010	77,260	95,384	104,278	122,465	141,004	154,488
	Ts	118.79	138.51	147.09	151.4	155.99	163.46	170.91	175.5	179.86	186.7
	F	***0.028***	***0.03***	***0.028***	***0.031***	***0.031***	***0.033***	***0.031***	***0.03***	***0.03***	***0.032***
	S	***4.1***	***7.36***	***9.7***	***13.38***	***16.12***	***20.26***	***20.4***	***22.19***	***25.16***	***28.15***
PRT	E	526	1066	1548	2113	2641	3138	3490	4156	4714	5198
	R	829	1571	2512	3328	4167	4871	5927	6736	7054	7650
	Ts	15.82	16.76	17.94	18.38	19.64	20.44	21.32	22.26	22.92	23.83
	F	***0.288***	***0.299***	***0.281***	***0.288***	***0.288***	***0.289***	***0.27***	***0.282***	***0.294***	***0.295***
	S	***33.25***	***63.6***	***86.29***	***114.96***	***134.47***	***153.52***	***163.7***	***186.7***	***205.67***	***218.13***
BPRT	E	714	1434	1999	2945	3596	4238	4940	6057	6386	7175
	R	712	1374	2173	3154	3518	4054	4841	5846	6416	6744
	Ts	15.76	16.6	17.21	18.35	19.05	20.18	20.9	21.64	22.38	23.61
	F	***0.417***	***0.425***	***0.386***	***0.412***	***0.422***	***0.422***	***0.417***	***0.437***	***0.414***	***0.429***
	S	***45.3***	***86.39***	***116.15***	***160.49***	***188.77***	***210.01***	***236.36***	***279.9***	***285.34***	***303.9***

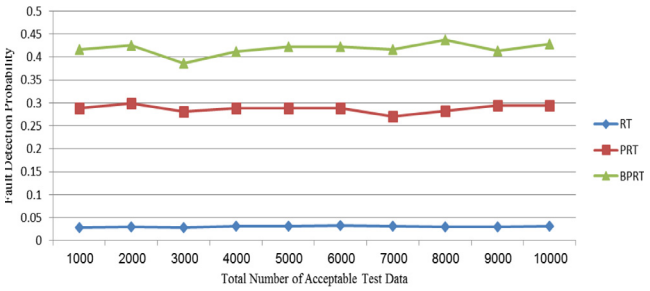

Figure 14 Fault detection probability chart of the program Middle with $N \geqslant 1000$.

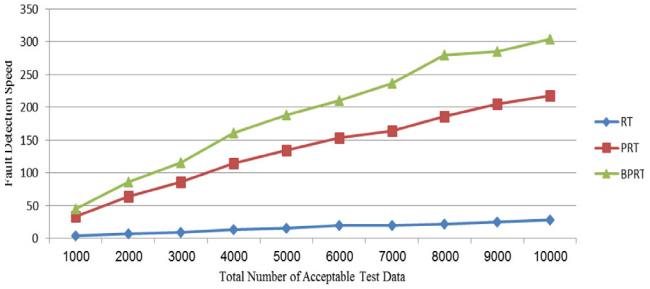

Figure 15 Fault detection speed chart of the program Middle with $N \geqslant 1000$.

number of acceptable test data are shown in Figs. 16 and 17, respectively.

With the experiments results of these methods about the program Tcas, a conclusion can be made that the fault detection probability of the BPRT method is about 4500 and 2.2 times more than that of the RT and PRT methods, respectively. It is also observed that fault detection speed of the BPRT method is about 277 and 1.6 times more than that of the RT and PRT methods, respectively.

5.6. Experiments results on program Totinfo

The Totinfo program is constructed of 7 functions, 2 arrays and 2 global variables. A path from InfoTbl function is selected to perform the testing. The selected path is the longest path of this function. This path contains nested loops and also conditional statements. Consider the experimental results on this path in Table 7. Due to the time consuming experiments regarding this path, the results have been recorded for $N = 100$–1000 with the step interval 100.

The fault detection probability of the aforementioned methods with the total number of acceptable test data has been shown in Fig. 18. In this experiment, the fault detection speed of the total number of acceptable test data has been shown in Fig. 19.

With the experiment results of these methods around the program Totinfo, a conclusion is made that the fault detection probability of the BPRT method is about 2700 and 1.7 times more than that of the RT and PRT methods, respectively. It is also observed that fault detection speed of the BPRT method is about 289 and 1.8 times more than that of the RT and PRT methods, respectively. Moreover, fault detection speed of the BPRT method increases, using a greater amount of test data, more than the other two methods. Nonetheless, outcomes from experiments on larger benchmarks need to be verified (see Figs. 20 and 21).

5.7. Experiments on the rate of faulty versions detection

There are 41 and 23 faulty versions of the programs Tcas and Totinfo, respectively. The majority of these versions contain only one error. Another criterion for the comparison is the abilities of different approaches to discover the faulty versions. Graphs of 20 and 21 show the number of Tcas and Totinfo faulty versions that are detected against the different numbers of test cases generated using the RT, PRT and BPRT methods. These results have been collected by running the algorithms for $N = 100$–1000 with the step interval 100.

As it can be observed, the number of faulty versions detected by our method is higher than that of the other two methods. Furthermore, these experiments show that the detection rate is higher at the first and this rate is reduced when the number of generated test cases is increased. This observation is more heighted in the experiments on Totinfo. It can be concluded, therefore, the number of versions that have been discovered in the early process of generating test cases is higher

Table 6 Experiments results of the program Tcas with $N \leqslant 1000$.

N		100	200	300	400	500	600	700	800	900	1000
RT	E	33	62	91	117	162	184	219	247	281	316
	R	1,721,637	3,470,225	5,399,979	6,737,013	8,731,610	10,473,727	11,587,160	13,562,658	15,297,654	17,101,046
	Ts	591.4	1205.7	1574.0	2470.9	2858.8	3486.3	3642.5	4015.6	4266.9	5080.6
	F	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***
	S	***0.06***	***0.05***	***0.06***	***0.05***	***0.06***	***0.05***	***0.06***	***0.06***	***0.07***	***0.06***
PRT	E	31	65	93	121	157	185	223	256	273	318
	R	734	1385	2091	2919	3417	4458	4711	5539	6230	7164
	Ts	3.8	6.9	7.8	12.9	16.7	17.5	20.2	21.6	25.3	31.2
	F	***0.037***	***0.041***	***0.039***	***0.036***	***0.04***	***0.037***	***0.041***	***0.04***	***0.038***	***0.039***
	S	***8.16***	***9.42***	***11.92***	***9.38***	***9.4***	***10.57***	***11.04***	***11.85***	***10.79***	***10.19***
BPRT	E	47	98	135	185	238	294	323	387	441	490
	R	466	864	1324	1866	2168	2813	3204	3520	3794	4516
	Ts	3.5	6.4	7.9	11.9	16.6	17.4	19.8	22.3	24.7	29.3
	F	***0.083***	***0.092***	***0.083***	***0.082***	***0.089***	***0.086***	***0.083***	***0.09***	***0.094***	***0.089***
	S	***13.43***	***15.31***	***17.09***	***15.55***	***14.34***	***16.9***	***16.31***	***17.35***	***17.85***	***16.72***

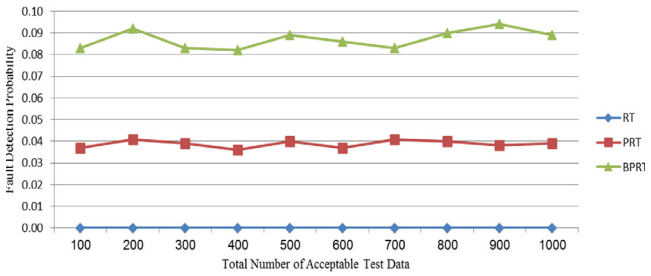

Figure 16 Fault detection probability chart of the program Tcas with $N \leqslant 1000$.

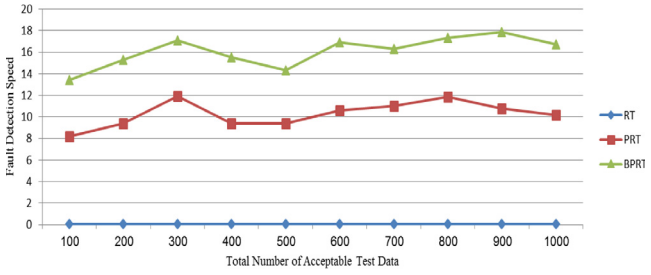

Figure 17 Fault detection speed chart of the program Tcas with $N \leqslant 1000$.

than the last ones, and perhaps this observation is closely relevant to the Pareto 80–20 principle. Further experiments in this direction could be to give more reliable results.

6. Threats to validity

This Section is dedicated to threat to validity in our studies, including external and internal validity. We use Turbo C++ 4.5 to implement our tools for test data generation. Threats to internal validity concern with possible errors in our implementations that could affect our finding. Nevertheless, we carefully checked most of our outcomes for decreasing these threats considerably. Also, the randomness of the proposed algorithm instead of the non-random can be a threat.

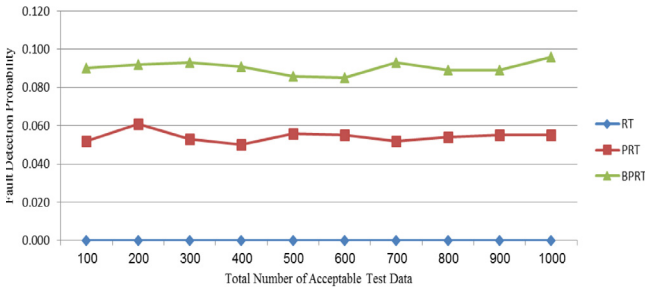

Figure 18 Fault detection probability chart of the program Totinfo with $N \leqslant 1000$.

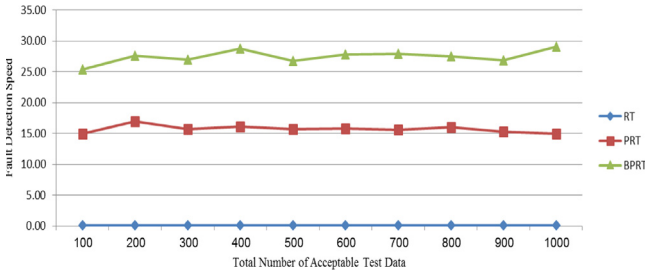

Figure 19 Fault detection speed chart of the program Totinfo with $N \leqslant 1000$.

The main threat to external validity is that our experiments are restricted to only five small or medium-sized programs. More experiments on larger programs may further strengthen the external validity of our findings. Further investigations of other programs in different programing languages would help generalize our results. Moreover, these programs were originally written in C, so they do not use object-oriented features such as inheritance, polymorphism and associations. Therefore, the results may not generalize our finding. Additionally, there is exactly one seeded fault in every Siemens program; in practice, programs contain much more complex error patterns.

Table 7 Experiments results of the program Totinfo with $N \leqslant 1000$.

N		100	200	300	400	500	600	700	800	900	1000
RT	E	25	55	74	98	125	157	183	199	230	242
	R	756,560	1,558,513	2,118,368	3,250,584	3,809,768	4,577,188	5,273,223	6,060,045	6,740,949	7,452,172
	Ts	265.2	542.6	782.6	1113.4	1307.9	1629.5	1840.2	2135.2	2347.1	2585.9
	F	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***	***0.000***
	S	***0.09***	***0.10***	***0.09***	***0.09***	***0.10***	***0.10***	***0.10***	***0.09***	***0.10***	***0.09***
PRT	E	24	56	77	98	127	149	176	202	222	247
	R	360	711	1152	1548	1767	2124	2664	2916	3168	3509
	Ts	1.6	3.3	4.9	6.1	8.1	9.4	11.3	12.6	14.5	16.5
	F	0.052	0.061	0.053	0.050	0.056	0.055	0.052	0.054	0.055	0.055
	S	15.00	16.97	15.71	16.07	15.68	15.85	15.58	16.03	15.31	14.97
BPRT	E	33	69	97	138	158	195	240	267	290	346
	R	267	550	747	1121	1340	1682	1871	2216	2349	2617
	Ts	1.3	2.5	3.6	4.8	5.9	7.0	8.6	9.7	10.8	11.9
	F	***0.090***	***0.092***	***0.093***	***0.091***	***0.086***	***0.085***	***0.093***	***0.089***	***0.089***	***0.096***
	S	***25.38***	***27.60***	***26.94***	***28.75***	***26.78***	***27.86***	***27.91***	***27.53***	***26.85***	***29.08***

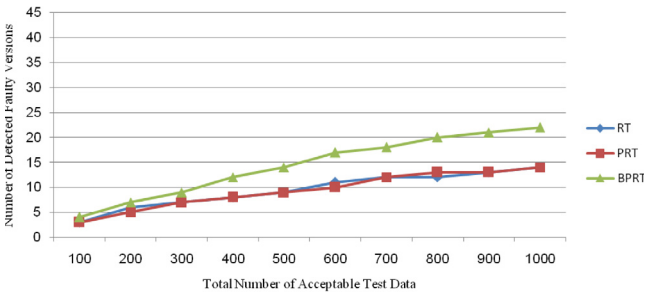

Figure 20 Number of faults detected in the program Tcas.

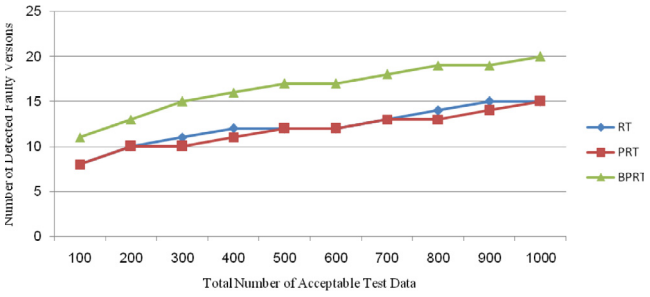

Figure 21 Number of faults detected in the program Totinfo.

7. Discussion and conclusions

In this paper a new approach was proposed for path-oriented test data generation. The main goal of the approach was to generate test data that lead to fault detection in short time. Most faults occur around the allowable area boundary and also at the boundary hypercuboids. Therefore, most test data are generated from the boundary hypercuboids. After explaining the proposed algorithm, it was evaluated and compared with other related works based on new and important criteria. To evaluate the interest rate of different methods to the fault detection, two criteria, (fault detection probability and fault detection speed) are defined. According to these criteria, the obtained BPRT fault detection chance of the generated test data is four times more than that of the PRT method. To evaluate the mentioned criteria more exactly, several experiments were done on five programs. On average, according to these experiments, the fault detection probability of the BPRT method obtained is about 1459 and 4.45 times more than that of the RT and PRT methods, respectively. The fault detection speed of this method is obtained about 129 times more than that of the RT method, on average, and 3.48 times more than that of the PRT method. Therefore according to the values obtained from the completed experiments, the important goals such as discovering more faults in less time, decreasing test costs, reducing wasted resources, increasing the fault detection speed and on time or even in time delivery of the product to the end-user were achieved. As for future works, extending the proposed approach so that having the support capability of all the operators and data structures such as bitwise operators, pointers and floating-point variables, will be proposed. Moreover, presenting a solution to solve the undesirable elimination of some boundary hypercuboids through the proposed consistency checking algorithm will improve the quality of the test process.

Acknowledgment

We are grateful to the unknown reviewers for their valuable comments. The comments significantly improved the paper and our approach.

References

Cai, K., Dong, Z., Liu, K., 2008. Software testing processes as a linear dynamic system. Inf. Sci. 178 (6), 1558–1597.

Hamlet, R., 1995. Software Quality, Software Process, and Software Testing. In: Proceedings of Advances in Computers, vol. 41, Elsevier, pp. 191–229.

Korel, B., 1990. Automated software test data generation. IEEE Trans. Softw. Eng. 16 (8), 870–879.

Ryu, S., Yi, K., 1999. Automatic test data generation for exceptions in first-order ML programs. Res. Program Anal. Syst., ROPAS Memo 1999–3.

Nirpal, P., Kale, K., 2011. Comparison of software test data for automatic path coverage using genetic algorithm. Int. J. Comput. Sci. Eng. Technol. 2 (2), 42–48.

Mansour, N., Salame, M., 2004. Data generation for path testing. J. Softw. Qual. 12 (2), 121–136.

Zhang, J., Xu, C., Wang, X., 2004. Path-oriented test data generation using symbolic execution and constraint solving techniques. In: International Conference on Software Engineering and Formal Methods, Beijing, China, pp. 242–250.

Hentenryck, P., Saraswat, V., Deville, Y., 1998. Design, implementation, and evaluation of the constraint language cc(fd). J. Logic Program. 37 (1–3), 139–164.

Offutt, A.J., Jin, Z., Pan, J., 1999. The dynamic domain reduction procedure for test data generation. Softw. Pract. Experience 29 (2), 167–193.

Chen, Y., Zhong, Y., 2008. Automatic path-oriented test data generation using a multi-population genetic algorithm. In: Proceedings of Fourth International Conference on Natural Computation (ICNC '08), Jinan, China, pp. 566–570.

Gotlieb, A., Petit, M., 2010. A uniform random test data generator for path testing. J. Syst. Softw. 83 (12), 2618–2626.

Zhao, R., Huang, Y., 2012. A path-oriented automatic random testing based on double constraint propagation. Int. J. Softw. Eng. Appl. 3 (2), 1–11.

Riaz Ahamed, S.S., 2009. Studying the feasibility and importance of software testing: an analysis. Int. J. Eng., Sci. Technol. 1 (3), 119–128.

Abreu, B.T., Martins, E., Sousa, F.L., 2005. Automatic test data generation for path testing using a new stochastic algorithm. In: Proceedings of 19th Brazilian Symposium on Software Engineering (SBES), pp. 247–262.

Salem, A.M., Rekab, K., Whittaker, J.A., 2004. Prediction of software failures through logistic regression. Inf. Softw. Technol. 46 (12), 781–789.

Edvardsson, J., 1999. A survey on automatic test data generation. In: Second Conference on Computer Science and Engineering, vol. 11, Linkoping, Sweden, pp. 21–28.

Alzabidi, M., Kumar, A., Shaligram, A.D., 2009. Automatic software structural testing by using evolutionary algorithms for test data generations. Int. J. Comput. Sci. Network Secur. 9 (4), 390–395.

Swathi, J.N., Sumaiya, T.I., Sangeetha, S., 2011. Minimal test case generation for effective program test using control structure methods and test effectiveness ratio. Int. J. Comput. Appl. 17 (3), 48–53.

Catelani, M., Ciani, L., Scarano, V.L., Bacioccola, A., 2011. Software automated testing: a solution to maximize the test plan coverage and to increase software reliability and quality in use. Comput. Stand. Interfaces 33 (2), 152–158.

Gong, D., Zhang, W., Yao, X., 2011. Evolutionary generation of test data for many paths coverage based on grouping. J. Syst. Softw. 84 (12), 2222–2233.

Boghdady, P.N., Badr, N.L., Hashem, M., Tolba, M.F., 2011. A proposed test case generation technique based on activity diagrams. Int. J. Eng. Technol. 11 (3), 37–57.

Myers, G.J., 2004. The art of software testing. In: John Wiley and Sons, second ed. (15), Hoboken, New Jersey, pp. 11–104.

Kosindrdecha, N., Daengdej, J., 2010. A test case generation process and technique. J. Softw. Eng. 4 (4), 265–287.

Xiao, M., El-Attar, M., Reformat, M., Miller, J., 2007. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. Empir. Softw. Eng. 12 (2), 183–239.

Pressman, R., 2003. Software Engineering a Practitioner's Approach, fifth ed. McGraw-Hill, New York, pp. 437–639.

Hierons, R.M., 2006. Avoiding coincidental correctness in boundary value analysis. ACM Trans. Softw. Eng. Method. (TOSEM) 15 (3), 227–241.

Chen, T.Y., Huang, D.H., Tse, T.H., Yang, Z., 2007. An innovative approach to tackling the boundary effect in adaptive random testing. In: Proceedings of the 40th Hawaii International Conference on System Sciences, Waikoloa, Big Island, HI, USA, pp. 262a.

Zhao, R., Li, Z., 2009. Boundary value testing using integrated circuit fault detection rule. In: Proceedings of the 2009 Testing: Academic and Industrial Conference – Practice and Research Techniques, IEEE Computer Society, Washington, DC, USA, pp. 3–12.

Ostrand, T., Weyuker, E., 2002. The distribution of faults in a large industrial software system. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). ACM Press, Roma, Italy, pp. 55–64.

Pareto, V., 1971. Manual of Political Economy. Translation of 1906 Edition, Augustus M. Kelley, New York.

Juran, J., 1951. Quality Control handbook. McGraw-Hill, New York.

Boehm, B., Basili, V., 2001. Software defect reduction top 10 list. IEEE Softw., IEEE Comput. Soc. 34 (1), 135–137.

Iqbal, M., Rizwan, M., 2009. Application of 80/20 rule in software engineering waterfall model. In: Proceedings of the 2nd International Conference on Information and Communication Technologies (ICICT), Karachi, pp. 223–228.

Rizwan, M., Iqbal, M., 2011. Application of 80/20 rule in software engineering rapid application development (RAD) model. In: Software Engineering and Computer Systems Conference, pp. 518–532.

Lipovetsky, S., 2009. Pareto 80/20 law: derivation via random partitioning. Int. J. Math. Educ. Sci. Technol. 40 (2), 271–277.

Pandey, V., Bairwa, A., Bhattacharya, S., 2013. Application of the Pareto principle in rapid application development model. Int. J. Eng. Technol. 5 (3), 2649–2654.

Kuo, C.S., Huang, C.Y., 2010. A study of applying the bounded generalized pareto distribution to the analysis of software fault distribution. In: Proceedings of the IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), Macao, China, pp. 611–615.

Kuo, C.S., Huang, C.Y., Luan, S.P., 2012. A study of using two-parameter generalized pareto model to analyze the fault distribution of open source software. In: Proceedings of the IEEE Sixth International Conference on Software Security and Reliability (SERE), Gaithersburg, MD, pp. 88–97.

Andersson, C., Runeson, P., 2007. A replicated quantitative analysis of fault distributions in complex software systems. IEEE Trans. Softw. Eng. 33 (5), 273–286.

Xiangyun, Z., Wenjing, L., 2011. Efficient testing model based on pareto distribution. In: Proceedings of the IEEE 3rd International Conference on Communication Software and Networks (ICCSN), Xi'an, China, pp. 314–316.

Fenton, N.E., Ohlsson, N., 2000. Quantitative analysis of faults and failures in a complex software system. IEEE Trans. Softw. Eng. 26 (8), 797–814.

Zhang, H., 2008. On the distribution of software faults. IEEE Trans. Softw. Eng. 34 (2), 301–302.

Gittens, M., Kim, Y., Godwin, D., 2005. The vital few versus the trivial many: examining the pareto principle for software. In: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC), vol. 1, Edinburgh, Scotland, UK, pp. 179–185.

McCabe, T.J., 1976. A software complexity measure. IEEE Trans. Softw. Eng. 2 (4), 308–320.

Ehmer-Khan, M., 2011. Different approaches to white box testing technique for finding errors. Int. J. Softw. Eng. Appl. 5 (3), 1–14.

King, J.C., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394.

Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R., 2007. The software model checker BLAST: applications to software engineering. Int. J. Softw. Tools Technol. Transfer 9 (5–6), 505–525.

Hutchins, M., Foster, H., Goradia, T., Ostrand, T., 1994. Experiments on the effectiveness of dataflow- and control flow-based test adequacy criteria. In: Proceedings of the 16th International Conference on Software Engineering, pp. 191–200.

Dwyer, M.B., Elbaum, S., Hatcliff, J., Rothermel, G., Do, H., Kinneer, A., 2012. Software-artifact infrastructure repository. Available at <http://sir.unl.edu/portal/index.php>, Last visited (Dec 2012).